Teo Hiltunen

C++ Robotics Library

Bachelor of Engineering

Spring 2018

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author: Teo Hiltunen

Title of thePublication: C++ robotics library

Degree title: Bachelor of Engineering, Information Technology

Keywords: C++, robotics, networking, Raspberry Pi

This thesis was built around the idea of creating a C++ robotics library for the Raspberry Pi. In its entirety, this project began in the fall 2017. For the duration of the project, new code was written and tested, all of which was published as open source in the internet.

In terms of documentation, the emphasis was on C++ programming. The final product contains many basic to intermediate difficulty level topics about writing good C++ code. This thesis aims to deliver a limited set of subjectively selected topics, presented through objective means. Based on the initial theory, there are samples of the library's code that demonstrate the use of the presented ideas.

The latter part of the thesis has a networking theme. It builds on top of the previous subjects, while introducing some of the basic concepts in network programming.

A total of three different libraries were created: *Net*, *Sync* and *GPIO*. The libraries do work, but their functionality has plenty to improve on. Creating these libraries has given some understanding on how library code should be written in practice. Creating a code library can be a long, arduous process. Active efforts to improve general programming skills helps writing better library code that others can understand.

TIIVISTELMÄ

Tekijä: Teo Hiltunen

Työn nimi: C++ robottikirjasto

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Avainsanat: C++, robotiikka, verkko, Raspberry Pi

Tämän opinnäytetyön tavoitteena oli luoda robotiikkakirjasto Raspberry Pi:lle, käyttäen C++-ohjelmointikieltä. Kokonaisuudessaan työ aloitettiin syksyllä 2017. Sen aikana kirjoitettiin ja testattiin uutta koodia, joka projektin lopussa julkaistiin vapaana lähdekoodina internetissä.

Dokumentointi painottui C++-ohjelmointiin. Viimeistelty työ sisältää joukon C++-aiheita, jotka soveltuvat niin kokeneille kuin kokemattomillekin ohjelmoijille. Opinnäytetyön dokumentaatio pyrkii tarjoamaan rajatun määrän subjektiivisesti valittuja aiheita, jotka esitetään objektiivisin menetelmin. Alustavan teorian jälkeen esitellään kirjaston koodia. Esitetyt ideat tulevat koodissa esiin käytännöllisestä näkökulmasta.

Jälkimmäisessä osassa on verkko-ohjelmointiin suuntaava teema. Verkko-ohjelmoinnin kannalta oleellisia perusteita käydään läpi, jonka lomassa sisältö rakentuu jo aikaisemmin todennetun tiedon varaan.

Jaottelusyistä työn lopptuloksena syntyi kolme erillistä kirjastoa: Net, Sync sekä GPIO. Niiden toimivuutta parannetaan opinnäytetyön jälkeisessä jatkokehityksessä. Opinnäytetyön aikana kertyi ymmärrystä ja käsitystä oman kirjaston luomiseen tarvittavasta tietotaidosta. Koodikirjaston luominen on pitkä prosessi, jonka oppimiseen voi myötävaikuttaa aktiivisella ohjelmointitaitojen parantelulla sekä tekemällä asiaan kuuluvia projekteja.

FOREWORDS

When I started studying at Kajaani University of Applied Sciences, I didn’t know much about progamming. I quickly developed a natural interest to it, and started learning lots. Learning at school wasn’t enough, so I did what I could outside the study hours. I participated in many memorable projects and got to know a lot of interesting people with different backgrounds. For this thesis I wanted to pass on some of the things that I’ve learned so far. I am very thankful for all of the support that I’ve gained from home, school, old friends and new ones.

TABLE OF CONTENTS

LIST OF TERMS

CPU	Acronym for Central Processing Unit. The CPU is the heart of the computer. It processes data coming from the input devices and sends it to the output devices. [1, p. 119]
Compile time	An action that occurs during the compilation of a program is said to happen during compile time.
GPIO	Acronym for General Purpose Input/Output. Way of communicating between the CPU and a peripheral is to use a GPIO pin. [2]
IDE	Acronym for Integrated Development Environment. IDEs provide the user with an interface that allows them to develop applications using a set of tools.
Library (computing)	A collection of reusable functionality that can be shared between multiple projects.
Peripheral (device)	An external device from the CPU’s perspective [3].
RPI	A common acronym for the Raspberry Pi.
Runtime	An action that occurs during the execution of a program is said to happen during runtime.
Serialization	Act of transferring data from one place to another. Commonly used when writing or reading from a data stream such as the contents of a file or a network stream.

1 INTRODUCTION

Recently, the use of virtualization has been on the rise. The word *server* is more relevant than ever. Data can be stored in a cloud, and software can be run separate from the client interface. To be able to build these network infrastructures, one must be competent in data serialization, understand large systems and abstract concepts. But before all else, a solid understanding of the programming language is required.

Creating a good C++ library requires good programming practices. A library is useless if the user cannot understand how to use it. And even then, the interface can be misleading. Eventually even the most seasoned programmer becomes a user of his or her own code, as it is impossible or trivial to remember all of the implementation details. There exists practically limitless options for writing clear, self-documenting code, each one with their benefits and drawbacks. As perfect programmers do not exist, there is always room for some improvement.

The objective of this thesis was to create a C++ library for programming applications that use peripheral devices. Target platforms were the Raspberry Pi and regular 64 bit desktop computers. Emphasis was on creating network tools that allowed the RPI unit to communicate with a remote desktop server application.

The thesis is divided into three major parts, each one subtly building on what was earlier established. Part one consists of an assortment of guidelines for writing better C++ code. The second part introduces the RPi and some basics for peripheral device programming. The final part has a networking theme.

2 WRITING GOOD C++

The programmer should be aware of the possibilities, limitations and the pitfalls of the language. C++ is a complex language, that is hard to master. For its difficulty, it offers equally powerful tools and control. Notably, C++ allows the direct management of memory, the most fundamental part of any program. The language is constantly evolving, and so the programmer is often required to adapt to new standards. In his book, *The C++ Programming language* [4, p.7], Bjarne Stroustrup – the original creator of C++ – says the following:

> *C++ is a language for someone who takes the task of programming seriously. Our civilization depends critically on software; it had better be quality software.*

This chapter covers a range of topics suitable for C++ programmers with varying levels of experience. The goal is to propose ideas that one can absorb into his or her own programming style. The claims are based on verifiable data as well as personal experience, in which case reasoning is provided.

2.1 Programming paradigms

A programming paradigm is a way of programming [5]. A programming paradigm is required for approaching the solution of a problem. Each problem involves a set of data that should be analyzed. The programming paradigms are the very fundamental way of how the programmer solves a problem in the code. There are numerous programming paradigms, some more used than the others. Some programming languages are better suited, or explicitly created around specific programming paradigms.

In the case of C++, there are at least three important ones that will be covered. Classes are and always have been a central aspect in C++. Because of this, object oriented design is covered. Functional programming is introduced to counter a flaw that emerges in some carelessly designed object oriented systems. Data oriented design is also covered, as C++ allows the direct memory control that helps to utilize this paradigm.

2.1.1 Object oriented programming

An object oriented approach is taught in most schools. It involves modeling the code after their real world counterparts. A class consists of a set of variables, the *attributes*, and a set of functions, the *methods* [4, p. 202]. For example, a class representing a Car contains data that is relevant to a car, and its methods are used to manipulate its state and to produce meaningful output in its context. Object oriented design is a good entry point for a new programmer, because one can better understand the used data and functionality through real life experiences and expectations.

2.1.2 Functional programming

Functional programming, or a functional approach addresses a flaw that comes with the usage of object oriented approach's class methods. Inside a class, a non-const method is free to modify the class' state and any accessible external state for that matter. Because of this, the method caller may accidentally end up modifying a state that was not meant to be modified. A dangerous, yet usual case is when the method modifies the state of a non-member variable. The consequences of modifying the state of the program without the awareness of the programmer can be unpredictable.

In functional programming the user provides the function with the inputs while the function provides the user with the outputs. There should be no side effects [6]. The advantage of this approach is that the user will know for certain, that nothing will be modified within the function. Ultimately the user decides what will be modified based on the function outputs. Because the function inputs and outputs are so explicitly defined, such functions are likely to be reusable.

Once initialized, a variable's state should not be modified. By following this rule, the functional approach opens new possibilities in the field of concurrent execution. If the state of a variable stays constant, then multiple threads won't encounter data races, situations where two or more threads are trying to acquire access to a segment in memory at the same time. [7]

2.1.3 Data oriented design

Data oriented design is fundamentally very different from the above. Data oriented programming aims to align data based on how it is accessed in the physical memory of the computer.

Whenever the CPU needs to access an address in memory, it first looks through an area of memory called the *cache*. If the requested memory resides in the cache, then the memory can be operated on. Otherwise, a *cache miss* has taken place. Upon a cache miss, the CPU needs to locate the requested memory address in the RAM section of memory and retrieve it into the cache. This retrieval process is very time consuming, and can take even ten times the amount of clock cycles compared to finding the memory in the cache. Retrieving the requested memory actually retrieves a large chunk of memory along with it. If the programmer is able to increase the likelihood of this memory chunk containing soon to be relevant data, then a number of cache misses may be avoided, resulting in a significant increase in performance. [8]

In his speech, Mike Acton argues that our mental model used for programming is misleading [8]. He gives an example of how the object oriented programming technique might try to deal with the programming of a chair (Figure 1). The three different types of chairs share a common interface, *the Chair*, when in fact all three are completely different. Acton suggests that following the principles set by the object oriented mind model will eventually lead into problems. This is because the programmer doesn't truly understand the underlying data that is being modified.

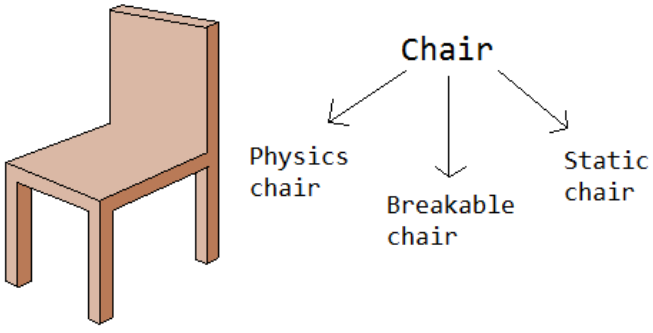

Figure 1. Chair example from Mike Acton's presentation [8]

2.1.4 Observations

It is possible to mix and match different programming paradigms and make use of each of their strengths. Object oriented design is well understood by many and convenient for fast prototyping. Functional programming can be used for clear, reusable functionality, that is likely to persist in the codebase for a long period of time. Data oriented design is an option, if the extra performance is required.

2.2 Clear user interfaces and coherent conventions

The interface of the library is the part that will be visible to the user. In most cases the user is interested in what it does, rather than how it does it. For this reason, it is important that the public interface is as clear as possible.

It is essential to define a set of rules which the programmer complies with from the start, a set of *coding conventions*. The standard and boost libraries are good examples of generic low-level libraries, which have well thought, clear namings, as well as a flexible use syntax [9] [10]. There are several recommendable characteristics for a clear interface.

Code should be built from generic elements that hold a single purpose. When a class or a function starts to expand, it is good idea to consider if it can be split into smaller entities.

Abstract classes can be very useful. An abstract class acts as an interface for specific functionality. An abstract class is a class that contains at least one *pure virtual function*, a function that needs to be implemented by the deriving class [4, p. 65]. It is recommended that an abstract class doesn't declare any variables, as the actual implementation is unknown to the interface.

Written code consists of declarations and definitions. Declarations usually reside in the *header files*. Header files can be included by the users of the library. As such, the contents of the header file require careful thinking. They will act as the user's reference, unless separate documentation is provided. Moreover, the programmer should consider what is the minimum amount of information required for display.

The implementation part is usually placed in a separate *source file*. Each source file is a *translation unit* of its own. Whenever a change in a header file is made, each source file including that header will have to be rebuilt. If a change is applied to the source file instead, only that file needs to be rebuilt. For this reason, code that is prone to change should always be placed in the source file, unless it absolutely needs to be visible to the other translation units.

A dependency is formed when a section of code relies on another, for example by calling a function or creating an instance of a class. These two pieces of code would usually reside in different translation units. Dependencies should be avoided if possible, especially *circular dependencies* where both sections of code rely on each other. When dependencies are used, they should follow hierarchical structures instead. As suggested in the header files paragraph, dependencies that require header files to be included are likely to increase build times. Dependencies can make applying a change in code difficult. However, one should bear in mind that internal dependencies are an unavoidable feature in any library.

Class inheritance should be minimal. The user is more likely to understand smaller hierarchies. Inheritance also introduce dependencies between the related classes. This is usually fine the first time when inheritance is used, but it can make future development troublesome. For example, creating an *object* base class with a three dimensional *position* property works for derived classes like *car* or *chair*, but not if a deriving class such as *health* is added. In this case the programmer would have to relocate the position property to something such as a *transform* component and then add the transform component to the chair and the car classes. If there were more classes that depended on the position being the object's property, this would mean substantial amount of work for the programmer.

Class and function names should be descriptive. Rather than having names that are fast to type, descriptive names can reduce the need for comments and make the code *self documenting*.

Any name can end up clashing with the user code. A name clash is a situation where two things that are fundamentally different share the same name. Name clashes can be prevented using a namespace. When the user wants to refer to a name inside a namespace, he or she must prefix the name with the name of the namespace separated by a double colon. Namespaces can be nested inside each other, and this can be useful for a growing library.

The user interface for file stream functionality in SpehsEngine [11] (Figure 2) can be found at the FileStream.h header file. The functions are encapsulated inside the spehs namespace. The comments above the functions explain the return value of each function. Some IDEs like Visual Studio display these comments when the function is referenced in the user code. Automated documentation generation tools like *doxygen* can generate clean documentation files based on formatted comments like these.

```
namespace spehs
{
	/** Returns true if directory exists at the given path */
	bool directoryExists(const std::string& path);

	/** Returns true if directory was created at the given path */
	bool createDirectory(const std::string& path);

	/** Returns true if file exists at the given path */
	bool fileExists(const std::string& path);

	/** Returns true if file exists and was removed successfully */
	bool removeFile(const std::string& path);
}
```

Figure 2. Extract from SpehsEngine [11] source code

2.2.1 Efficiency and readability

Generally, there could be considered to be at least two types of efficiency: write efficiency and run-time efficiency. Efficient writing means writing code at a fast pace. This could be achieved by shortening variable names or by using the *auto* keyword. If this kind of mentality is used, it is very important to revise the code once completed, so that only readable code remains for maintenance. The use of auto keyword is not advicable, as it makes the code hard to read for someone unfamiliar with the code.

Run-time efficiency is mostly needed for heavy algorithms and tasks, such as rendering graphics on the screen in a game. With platforms that possess limited processing power – such as a small microcontroller – any form of efficiency is desirable. Compilers automatically do optimizations on the source code. As an example, functions are inlined and values are calculated at compile-time [12]. All in all, the modern compiler optimizations can get very complicated. It is good to acknowledge their existence, and the fact that they can do some of the work for you. The programmer should concentrate on writing readable, self documenting code rather than performing micro-optimizations that make the code harder to read.

2.2.2 Pointers, references and assumptions

When writing the parameter types for a function, the programmer has two options: pass by value, or pass by reference. Passing by reference is recommendable, as it usually involves less copying. It also reduces compilation times because the function declaration doesn't need to be bundled together with an include to the file declaring the parameter type. In such cases a forward declaration will be sufficient.

Programming without any assumptions would be impossible. If a pointer points to something other than null, it should always be assumed to be valid in the context. A reference should always reference something, and never be initialized by dereferencing null. All reference variables must be initialized upon creation, and their reference value stays constant for the duration of their lifetime [13, References and Safety].

In Figure 3, the programmer first makes a call to the getTrasnsformPtr function that returns a pointer to a transform object. Because the return type is a pointer, it indicates that the returned value might point to null. If the return value is null, it means that the function failed to obtain the transform object or returned null for some other purpose. In any case, the programmer must prepare for the function returning null. The getTransformRef function returns a reference type which is guaranteed to reference a valid transform object. No further validation is needed, the reference is ready to be used.

```
Transform *transformPtr = getTransformPtr();
if (transformPtr)
{
	//Do something with transformPtr
}

Transform &transformRef = getTransformRef();
//Do something with transformRef
```

Figure 3. Using a pointer and a reference

2.2.3 Const qualifier

The const type qualifier is used to indicate that the state of an object stays constant during its lifetime. On the opposite, a *mutable* object can change its state at any given time. A mutable object can be given the const qualifier. Once an object has the const qualifier, it isn't meant to be removed, although this is possible.

Learning how to use const is crucial for creating clear, compatible user interfaces. It should be considered an universally understood way of documenting the code. A const qualifier instantly tells the user something about the nature of a variable or function (Figure 4).

```
void translate(
	Transform& t, // Mutable, the function must be mutating this object
	const vec3& v); // Const, the function will not mutate this object
```

Figure 4. The programmer can extract information based on the const qualifier in the function parameter types

The common problem with inexperienced programmers is the fact that code that hasn't been const qualified propagates to its surroundings (Figure 5). This causes a chain reaction where the programmer can easily give in and remove the const qualifier, further worsening the situation. The only solution at this point is to fix all of the related code to use the const qualifier.

```
class Projectile
{
	vec3 getDirection() const;
	float getVelocity(); // 1. The programmer forgot to add const here

	vec3 getMovementVector(const Time deltaTime) // 3. should be const!
	{
		// 2. Used method getVelocity() is not marked as const!
		// -> therefore this function is assumed to mutate the object
		return getDirection() * getVelocity() * deltaTime;
	}
};
```

Figure 5. A usual case where the const incorrectess propagates

When writing const into the code, it modifies the type on the left. The term *east const* is sometimes used. If there is no type on the left side, it is applied to the right side. Interestingly, due to old conventions, writing code like this is discouraged by many, such as Bjarne Stroustrup himself. Instead, we are encouraged to write code using the *conventional const notation* (Figure 6). [S, East const]

```
class ConstNotationExample
{
private:
	const bool conventionalConstNotation = true;//recommended
	bool const eastConstNotation = false;//valid syntax, but not recommended
public:
	//However, east const notation must be applied to member functions
	bool getEastConstNotationValue() const { return eastConstNotation; }
};
```

Figure 6. Different const notation examples

2.2.4 Strongly typed and strong types

C++ is a *strongly typed* language. It forces the user determine the used data types directly (Figure 7). When the programmer declares a variable, he or she must first declare its type. This process may seem tedious, but it can ultimately be used for the programmer's advantage. This forces the programmer to understand the managed data better, as he or she must actively make consious efforts when choosing the used types. Some type related errors can be catched during the compile time. For an experienced programmer, the written code will also provide better readability, as the used data types can clearly been seen.

```
int a = 1;
int b = 2;
int c = a / b; // = 0
float d = (float)a / (float)b; // = 0.5f
float e = a / (float)b; // = ? compiler will likely cast 'a' into float
```

Figure 7. Type related arithmetic problems are common among beginners

All data consists of the fundamental C++ data types or user defined types, which correspondingly consist of other user defined types or ultimately of the fundamental types [4, p. 138]. For any fundamental or user defined type, the programmer can add a type alias, using the *typdef* specifier. For example, the type define specifier can be used to give an interface clearer types (Figure 8). A recursive usage of type define may result in code that is harder to read and to maintain, so it is better to only use it on types that reoccur often and are central to the context.

```
typedef unsigned ObjectId;
void getObject(const ObjectId id);
```

Figure 8. The getObject function's id parameter becomes more verbal when a type defined type is used

A similar term, *strong type* is related. Strong types can be fundamental values, wrapped inside user defined classes. In his blog, Christian Boccara describes the use of strong types as an expressive way to make the code more error-prone and force the right order of arguments at the call site [15].

Code in figure 9 forces the user to explicitly pass a value of type ObjectId. As with a mere type define like in the case of Figure 8, the compiler could allow the use of any unsigned variable as a valid argument. However, ObjectId and PlayerId are still of the same type, and may potentially get mixed.

```
struct StrongInt
{
	int value;
};
typedef StrongInt ObjectId;
typedef StrongInt PlayerId;
Object* getObject(const ObjectId id);
Player* getPlayer(const PlayerId id);
```

Figure 9. Strong type example

Code in figure 10 declares ObjectId and PlayerId as different types. If the user accidentally tries to pass an ObjectId into the getPlayer function, the compiler will issue an error.

```
struct StrongInt
{
	int value;
};
struct ObjectId : public StrongInt {};
struct PlayerId : public StrongInt {};
Object* getObject(const ObjectId id);
Player* getPlayer(const PlayerId id);
```

Figure 10. Strong type example improved

Declaring new strong types can seem laborous. A preprocessor definition can be used to shorten the declaration syntax (Figure 11).

```
#define STRONG_INT(Name) struct Name : public StrongInt {};
STRONG_INT(ObjectId);
STRONG_INT(PlayerId);
```

Figure 11. Strong type preprocessor macro

2.3 Generic programming

Generic programming, or in C++'s case *template* programming is a technique to generalize certain parts of the code. An easy way to find a place to use a template is when the programmer notices that a specific structure of code repeats itself, but uses different types or type combinations. In template programming a body of a function, or the implementation of a class, is written with the intention of placing unspecified types or varying values into the template definition. The programmer then uses the template by providing it with some template parameters. When the code is compiled, the compiler instantiates the template code with those parameters, and compiles it. [16]

It is common tradition to first learn about templates that use typenames. However, templates can also be provided with values that are compile time constant. For example, an integer in the form of the number five. This can be useful for many cases, for example using an integer template parameter to specify the fixed size of a container (Figure 12). By doing this, the programmer can avoid doing an expensive dynamic memory allocation, and also have the flexibility of static sized arrays in a container class.

```
template<typename T, int size>
struct Container
{
	T data[size];
};

Container<int, 5> fiveInts { 0, 1, 2, 3, 4 };
```

Figure 12. A container that contains a specified quantity of speficied type is instantiated with the values 0, 1, 2, 3 and 4

As a final note, it is good to acknowledge that template heavy code can significantly lengthen build times [4, p. 697]. This is happens since the template code mostly resides in the header files, rather than source files. Not only are the header files more prone to change, but they also contain more dependency includes, since the template definitions may induce more required dependencies.

2.4 Utilizing preprocessor definitions

The development environment should have an option to switch between different configurations per platform, and the platform itself. This is automatically set up by some IDEs like Visual Studio. The configurations created by Visual Studio are *debug* and *release*.

The debug configuration prefers faster compile times and keeping the built machine instructions close to the original, human readable source code. By doing this, the program will be easier to debug, as the debugger will be able to provide more information that relates to the code written by the programmer. In the ideal case the debugger will be able to step from line to line without interruptions, and keeping track of nearby variable values.

Full compiler optimizations are usually run for the release version. After the optimizations, the produced executable code no longer maps to the written code. Debugging during the release configuration can therefore be harder or seemingly impossible.

There should be a preprocessor definition for deducing the active configuration. The definition can then be used to add code behind the preprocessor directives. Visual Studio defines NDEBUG in the release configuration, and _DEBUG in the debug configuration.

The assert technique (Figure 13) is commonly used to catch bugs in the debug configuration. Building with the release configuration is set to generate no code for optimization reasons.

```
#ifdef NDEBUG
#define ASSERT(_file) ((void)0) // Generates no code in NDEBUG (release)
#else
#define ASSERT(expression) { if (!(expression)) log::error("ASSERT failed."); }
#endif

       // A section in code:
       ASSERT(sanityCheck());// A sanity check doesn't need to be run in release
```

Figure 13. Assert definition

C++ is a widely supported language. The same library or project code may be desired to run on many different platforms. This introduces an issue, when some parts of the code are incompatible with a certain platform. For this reason, it is advisable to add a preprocessor definition for the currently active platform (Figure 14). All future code would then be considered with this in mind.

```
#define PLATFORM_WINDOWS 1
#define ACTIVE_PLATFORM PLATFORM_WINDOWS

void createDirectory(const char *path)
{
#if ACTIVE_PLATFORM == PLATFORM_WINDOWS
       //Windows specific directory creation code
#else
#error Platform implementation missing!
#endif
}
```

Figure 14. File and directory management is usually platform dependent code

2.5 Memory allocation

There are two types of runtime memory allocation: stack allocation and dynamic allocation. Stack is an area of memory that can be traversed linearly in two directions. When an object is allocated, it is allocated at the top of the stack. Stack allocated objects always get deallocated in the reverse order where they were allocated (Figure 15). This kind of a stack structure is sometimes referred to as *last in first out*. Stack allocation is fast and the object lifetime is well defined. The downside of stack allocation is the inability to allocate quantities that are unknown at compile time, such as dynamic size containers. Also, the stack size is somewhat limited and should not be used for large objects such as containers or buffers. [17]

```
{
        int a; // allocate a
        {
                int b; // allocate b
                {
                        int c; // allocate c
                } // deallocate c
        } // deallocate b
} // deallocate a
```

Figure 15. Stack allocation and deallocation demonstrated

Dynamic allocation happens using the *new* operator or the *malloc* function [17]. The allocated memory resides in the segment of memory called the *heap* [17]. Dynamic allocation should be used when the object size is varying, such as a container that needs to grow once it has used all of its allocated memory. All dynamic memory must be manually freed by the programmer, using the *delete* operator or the *free* function.

A source that allocates dynamic memory is said to own the memory. If the memory owner is deallocated without freeing the memory, the program has started to *leak* memory. The leaked memory will remain to preoccupy space in the program memory space. The danger being that the consecutive memory leaks will eventually end up consuming all of the available program memory, and stopping the program from working.

2.6 Multithreading

Modern CPUs have multiple cores, and are able to execute instructions in parallel [7]. For this reason, multithreading programs have become more important in the recent years.

A single threaded program has all of the control over its surrounding memory state. There is no danger of another thread mischievously modifying an area in the memory. When a program launches a new thread, the active threads running in different CPU cores are in danger of causing a data race with each other. This means, that two or more threads are using the same segments in memory at the same time. Such behaviour is undefined, and usually yields unexpected results. In the worst case, multithreading bugs can go unnoticed for a long time, until a change in some part of the code invokes this kind of a time critical bug.

Designing efficient and trouble-free multithreaded programs is hard and unintuitive. It is recommended to avoid multithreading if there exists a reasonable alternative. When using multiple threads, mutual exclusion or other similar technique must be used to prevent data races from happening. Mutual exclusion introduces the concept of mutex objects. Mutexes are objects that can be used to prevent multiple threads from accessing the same memory areas. A mutex object must be locked before accessing an appointed area of memory, for example a variable. The programmer decides which mutex object protects which variables. When the thread no longer needs the variable, the mutex lock must be unlocked. In case that another thread has already locked the mutex, the other thread will have to wait to acquire the lock, effectively blocking its execution.

Figure 16 displays an example of mutex usage. One can notice that there is nothing that prompts the user for using the mutex when accessing the variable, except the careful documentation that the programmer should write when writing multithreaded code. There are many things that can go wrong if the user negletcs the safety principles of multithreading. Most, if not all of those errors go unnoticed by the build process.

```
/*
   The programmer has decided that this mutable variable “var” will be accessed
   from multiple threads, potentially at the same time. Therefore, it must be
   mutex protected and falls under the protection of the “varMutex” object.
*/
int var = 5;
std::mutex varMutex;

int incrementAndReturnVar()
{
	varMutex.lock();
	const int value = ++var;
	varMutex.unlock();
	return value;
}
```

Figure 16. Mutex example

3 THE RASPBERRY PI AND GPIO PIN CONTROL

This chapter gives a brief explanation on the functionality of the Raspberry Pi and its GPIO pins. It also explains what functionality was needed for the robotics library, and how it was added.

3.1 Raspberry Pi

Raspberry Pi is commonly abbreviated as RPI, Raspi, or just the Pi. It is a small device, roughly the size of a credit card. Compared to its size, the RPI has a lot of memory and processing power. The RPI model 3 has four CPU cores with the clock speed of 1.2 GHz, 1 GB of RAM, 40 general purpose input/output pins and a built-in wifi module. The RPI is a popular platform for many of the do-it-yourself hobbyist. [18]

For comparison, an Arduino Uno microcontroller costs roughly half as much as a RPi. Uno has a single core that runs with the clock speed of 16 MHz. The difference in clock speed is substantial, but for many projects even an Uno will be sufficient. Memorywise there is only 32 KB of flash memory, 2 KB of SRAM and 1 KB of EEPROM. Again, the numbers for the RPi are much greater. [19]

The RPI uses its own operating system, Raspbian. It is a Debian based operating system that is free to download from their official website. [20]

3.2 Visual Studio for Linux Development extension

Visual Studio for Linux development was used for the development of the the library. It is a free extension available for the Visual Studio 2015 and 2017. The extension allows the user to write code on his or her desktop computer in Visual Studio. The extension then connects to a remote Linux system. Upon building, it copies the needed source code files to the remote system. It also prepares the needed commands and arguments that match the settings defined in the Visual Studio environment. The compilation happens remotely on the linux machine, so this is not a cross compiler. [21]

During the development of the library, various bugs were encountered while using this extension. The extension had difficulty in outputting readable error messages (Figure 17). Many times, the extension had failed to detect what translation units needed to be rebuilt when changes were made. In these cases a full rebuild was required. This slowed down the development process significantly, especially when the overall size of the robotics library growed. Another category of bugs involves the extension preparing incorrect command arguments for the remote system's compiler. These can be hard to catch as the programmer has to dive into the build output window in full verbose mode and compare the argument chain to whatever settings were defined in the project properties. Despite these difficulties, the extension was able to perform at times.

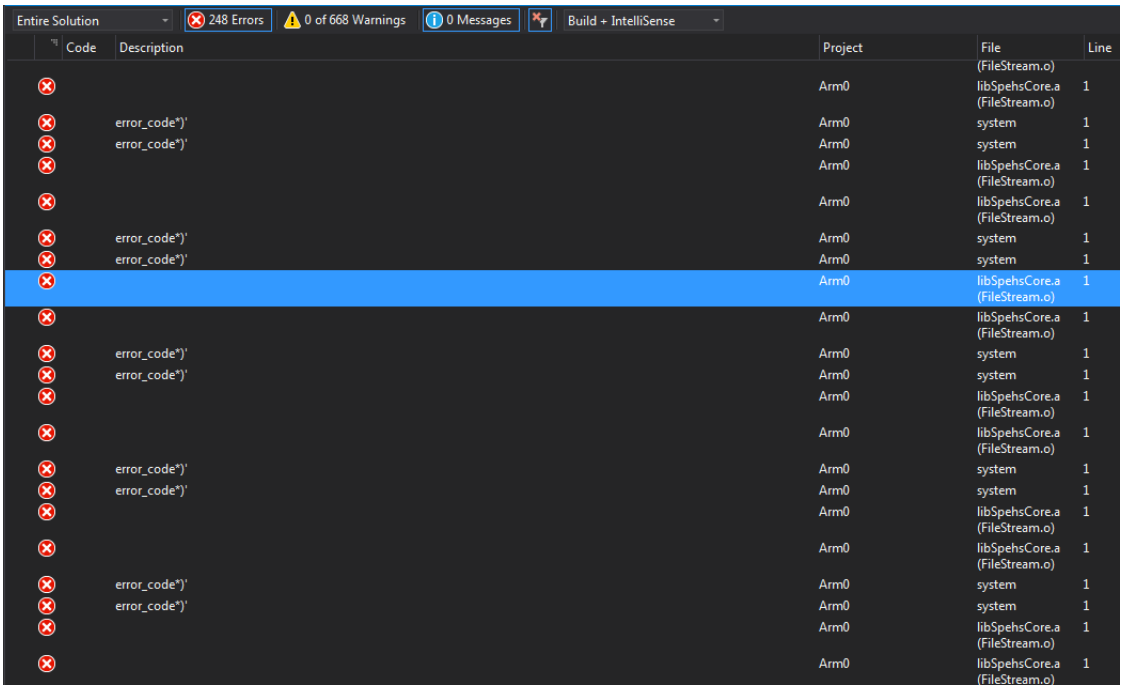

Figure 17. The error output received from the remote system is usually unreadable

3.3 GPIO pins

GPIO stands for general purpose input/output. Microcontroller GPIO pins are used to transmit information between the central computer and a peripheral device. They are generally configured in either of two ways: input or output. Input means, that the computer will be reading data from the pin, where as outputting is about writing data to the pin. [3]

The robotics library uses the bcm2835 library for controlling the GPIO pins. It provides all of the needed functionality for reading and writing into the pins. First tests were made by trying out different simple peripherals. Once the basic idea of the workflow was established, wrapper functions for pin control were added (Figure 18). The wrapper functions force the user to use the library's own strong types: Pin, PinState and PinMode. A wrapper function takes the cleanly formatted user input and calls the underlying bcm2835 implementation (Figure 19).

```
    void enable(const Pin pin);
    void disable(const Pin pin);
    void write(const Pin pin, const PinState pinState);

    PinState read(const Pin pin);
    spehs::time::Time pulseIn(const Pin pin, const PinState pinState, const
        spehs::time::Time timeout = 0);

    void setPinMode(const Pin pin, const PinMode mode);
    void setPinAsInput(const Pin pin);
    void setPinAsOutput(const Pin pin);
    Pin getPinNumberAsEnum(const unsigned number);
    unsigned getPinEnumAsNumber(const Pin pin);
```

Figure 18. The library's gpio pin control wrapper function interface

```
void setPinMode(const Pin pin, const PinMode mode)
{
#ifdef CODEX_GPIO
       if (mode == PinMode::output)
              bcm2835_gpio_fsel(pin, BCM2835_GPIO_FSEL_OUTP);
       else if (mode == PinMode::input)
              bcm2835_gpio_fsel(pin, BCM2835_GPIO_FSEL_INPT);
       else
              spehs::log::error("setPinMode() error. Invalid pin mode.");
#endif
}
```

Figure 19. The wrapper implementation calls the bcm library implementation

3.4 Peripheral devices

External devices, or *peripherals*, are like tools for the CPU to work on. Motors, sensors and essentially any input or output devices are peripheral devices. Their existence is crucial in most embedded systems, as they are the components that let the CPU to communicate with the outside world.

3.5 Servo motors

Servo motors usually require one GPIO pin that will be used to control the rotation. In this case the pin is referred to as the *signal pin*. The CPU sets the signal pin into *high* state for a set amount of time, then returns it to the *low* state (Figure 20). The amount of time spent in the high state determines the angle which the servo will try to approach. Servos usually map to the following range: 1 millisecond equals the lowest rotation, while 2 milliseconds equals the highest rotation. This mapping range is not universal, and it can vary based on the manufacturer.

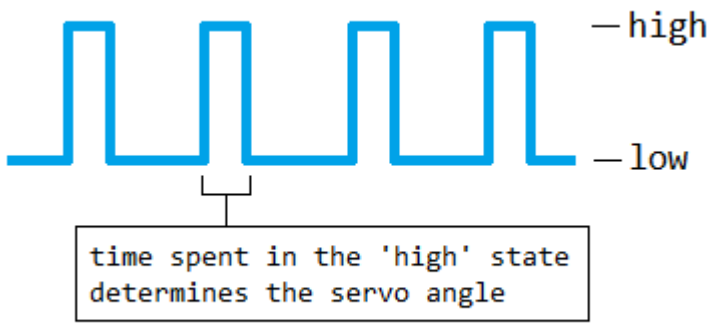

Figure 20. Servo signal pin signal example

3.6 Programming the servo class

The servo was the first peripheral device to be added to the library. An object oriented approach was used (Figure 21).

```
class Servo
{
public:
	Servo();
	~Servo();

	void setPin(const gpio::Pin pin);
	void mapPosition(const time::TimeType minPulseWidth,
		const time::TimeType maxPulseWidth);

	void run();
	void stop();
	void setPosition(const unsigned char position);
	gpio::Pin getPin() const;
	unsigned char getPosition() const;
	bool isRunning() const;

private:
	void runLoop();

	mutable std::recursive_mutex mutex;
	gpio::Pin pin;
	unsigned char position;
	time::TimeType minPulseWidth;
	time::TimeType maxPulseWidth;
	std::thread* runThread;
	bool keepRunning;
};
```

Figure 21. Servo class declaration

Before taken into use, the servo instance needs to be assigned with a signal pin. Afterwards the run method needs to be called. This launches an additional thread that starts controlling the signal pin based on the current position and the position mapping.

The position value is an unsigned char, which is typically an unsigned 8 bit integer. The value of 0 represents the lowest angle the servo can reach, and in the 8 bit case 255 being the highest. The default minimum high signal width of 1 millisecond and the maximum width of 2 milliseconds can be altered with the mapPosition method. As so, any position value between the min and the max positions get interpolated according to the min and max signal widths. For example, a position value of 127 would account for the signal width of roughly 1.5 milliseconds by default.

4 NETWORKING

Networking is an important aspect in many of today's systems. When two separate programs need intercommunication, networking is to be considered. It is an essential challenge when considering how programs are built. Programs consist of executable code and a memory space containing the variables. Two programs, different or the same, need a set of common rules, or a *protocol* to be able to understand each other and function properly.

Before the need for a protocol, the participating parties must first have some kind of a physical connection, such as an RJ-45 cable that connect the computers into a common network. A networking library such as boost's asio can handle the all of the low level protocols for establishing a reliable TCP connection between the two programs. Once a TCP based connection is established, the programs can start sending packets of data to each other. For this step the programmer needs to come up with a protocol so that the programs can understand the contents of each data packet.

4.1 Data serialization

In computing, storing and sending data requires the data to go through a process called serialization. Data serialization is a two step process: serialization and deserialization. The first one manipulates memory into a storeable/sendable format, where as deserialization is used to unpack this format back into its original state. It is needed, because data structures shouldn’t be directly copied from one section of memory to another. While this may work in some cases, the problem is that a direct copy cannot account byte endianness, pointers and reference values, or platform dependent implementations such as integer widths.

The C++ programming language does not provide an universally applicable feature for data serialization, so handling serialization usually falls into the consideration of the programmer. Serialization requires careful planning from the programmer, but it is usually a mechanical, straightforward task.

Figure 22 presents an example of how the given class *Data* could be serialized. Deserialization of the member attributes happens in the same order as serialization. The << and >> operators correspondingly write and read values to and from the stream. During serialization, data is written to the stream. During deserialization, data is read from the stream. It is recommended for the programmer to consider whether the serialization interface uses the const qualifier for the serialization step.

```
struct Data
{
	static const uint64_t version = 1;
	bool serialize(Stream &stream) const
	{
		stream << version;
		stream << i;
		stream << s;
		return true;
	}
	bool deserialize(Stream &stream)
	{
		size_t writtenVersion;
		stream >> writtenVersion;
		if (writtenVersion < version)
			return false;
		stream >> i;
		stream >> s;
		return true;
	}
	int32_t i;
	uint16_t s;
};
```

Figure 22. Serialization example

4.2 Types in serialization

In figure 22, serialization method, the programmer writes a sequence of variables into the stream: a size_t, an int and a float. The deserialization process assumes that the first 64 bits will be used for a variable of type uint64_t, followed by 32 bits for an int32_t and 16 bits for a short uint16_t.

The fundamental thought is the programmer's assumption that the next set number of bytes in the stream will be used to reconstruct a known type T.

Sometimes the serialized data is an array of element with a varying size, such as a string. In this case there are two common solutions. Either write the size of the array at the start, or a null terminator at the end (Figure 23).

```
void serialize_choice_A(Stream &stream, const String &string)
{
	stream << string.size();
	for (size_t i = 0; i < string.size(); i++)
		stream << string[13];
}

void serialize_ choice_B(Stream &stream, const String &string)
{
	for (size_t i = 0; i < string.size(); i++)
		stream << string[13];
	const char null = 0;
	stream << null;
}
```

Figure 23. Dynamic size array serialization

When serializing polymorphic classes (Figure 24), the programmer must provide a mechanism to deduct the type of the class located next in the streamThis can easily be achieved by providing a value representing the type (Figure 25). An enumerated value can be used. Before each polymorphic class write, the type value is written. When reading the stream, the type value is read first and based on that the programmer can allocate a dynamic object of the desired type. The dynamic allocation mechanism must also be provided by the programmer.

```
struct A
{
	virtual bool serialize(Stream &stream) const;
	virtual bool deserialize(Stream &stream);
};

struct B : public A
{
	bool serialize(Stream &stream) const override;
	bool deserialize(Stream &stream) override;
};

struct C : public B
{
	bool serialize(Stream &stream) const override;
	bool deserialize(Stream &stream) override;
};
```

Figure 24. Polymorphic serialization methods

```
enum class Type
{
	A, B, C
};

A *allocate(const Type type)
{
	switch (type)
	{
	default: return nullptr;
	case Type::A: return new A();
	case Type::B: return new B();
	case Type::C: return new C();
	}
}
```

Figure 25. The needed type’s size is unknown during compile time, so the allocation must be dynamic

4.3 High level TCP socket class

Boost asio network library was used for sending data packets over the network. However, its use is somewhat time consuming and error-prone. The library was extended with a *TCPSocket* class.

The socket class adds an additional custom protocol layer on top of the TCP. This custom protocol contains an additional packet data header. The header defines whether the data contents are user defined data or protocol specific data such as the handshake or disconnection. User data is passed on to a handler function, while protocol data is processed by the socket without the need to notify the user.

The downside of using this protocol is that it can only be used with parties that also understand the protocol. But for the scope of this project, its functionality is sufficient.

When the socket is connected (Figure 26), it starts another thread that upkeeps the connection and receives the other endpoint's messages. Any valid incoming message is placed in a queue, ready to be processed. The received messages can be processed any time by calling the socket's *update* method. By storing the messages in the queue, the user doesn't have to be aware of the underlying multithreading, as in, the messages are not processed from another thread upon receiving.

```
codex::IOService ioService;
codex::SocketTCP socket(ioService);
const codex::protocol::Endpoint endpoint("192.168.0.0", 0);
if (socket.connect(endpoint))
{
	spehs::log::info("connected successfully");
}
```

Figure 26. Instantiating and connecting a SocketTCP socket is easy

A server will use the socket's startAccepting method. It places the socket in a receiving state where it will listen for an incoming connection. Figure 27 displays an example usage of this. The startAccepting method has launched another thread so that the main thread is free to do other tasks while awaiting for the incoming connection. In this example the main thread will keep calling the update method until the socket has finished accepting.

```
void onAccept(SocketTCP& socket)
{
	//Do something on accept
}

…

socket.startAccepting(port, std::bind(&onAccept, std::placeholders::_1));
while (socket.isAccepting())
{
	socket.update();
}
```

Figure 27. Socket is awaiting for an incoming connection, *accepting*

4.4 Write/read buffer classes

Managing serialized data is an important aspect in any library that concerns networks. For the sake of exercise, write and read buffers were created for the library. Both buffers were proposed to be able to write and read data in the right byte endianness format. Byte endianness is the order that the CPU reads the bytes of a multi-byte variable (Figure 28). Byte endianness must be considered if a message is sent through a network, or a file is meant to be readable on another machine.

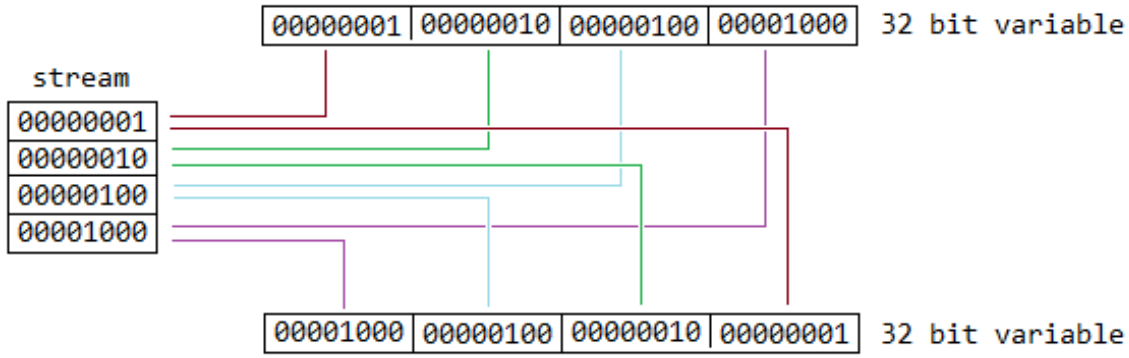

Figure 28. Two of the most common endianness types are *big* and *little* endian

The write buffer extends as data is written to it. The read buffer is constructed from a block of memory and doesn't own the underlying data, thus there is less copying involved.

C++ templating and SFINAE was used to create a nice use syntax for the classes (Figures 29, 30). If the passed argument type for the read and write functions is a class type, it needs to have read/write methods that know how to serialize and deserialize the class contents. For fundamental types, the buffer simply writes and reades according to the network byte order.

```
protocol::WriteBuffer buffer;
buffer.write(pin);
buffer.write(minPulseWidth);
```

Figure 29. Write buffer use syntax

```
protocol::ReadBuffer buffer(packets[13]->data(),packets[13]->size());
gpio::Pin pin;
buffer.read(pin);
spehs::time::Time minPulseWidth;
buffer.read(minPulseWidth);
```

Figure 30. Read buffer use syntax

4.5 Matchmaking server

Debugging two applications at once required them to know of each other's endpoints, which changed from time to time. This proved to be tedious, so an automated matchmaking tool was created. Another protocol layer was added for this purpose. The matchmaking protocol was dependent on the socket layer protocol, but not the other way around. This way the matchmaking layer could be removed at any point if so desired.

Once again, focus was on having a simple syntax for the end user. To run an ”Aria” matchmaking server, all that the user needed to provide was a local port number for the endpoint (Figure 31).

```
    codex::aria::Server aria;
    aria.start(port);
    while (aria.isRunning()) {}
```

Figure 31. Matchmaking server initialization

Figure 32 demonstrates the required client code. On the client side, things are a bit more complicated, but not too much. The user has to provide a socket that will be used for the connection, as well as know the endpoint of the matchmaking server. In addition, the service requires a name and a counterpart name to be provided for the match to happen.

```
codex::IOService ioService;
codex::SocketTCP socket(ioService);
const codex::protocol::PortType myPort = 41623;
codex::aria::Connector connector(socket, "myName", "counterpartName", myPort);
const codex::protocol::PortType remotePort = codex::protocol::defaultAriaPort;
codex::protocol::Endpoint endpoint("192.168.10.51", remotePort);
if (connector.enter(endpoint)))
    spehs::log::info("yay!");
else
    spehs::log::info("nay!");
```

Figure 32. Connecting the client to the server

4.6 Remotely controlled car robot

This was the first functional robot built with the library. It utilized the TCP socket class, the servo class and the DCMotorController class. The RPI was mounted into a small car frame, that had a DC motor attached to its rear wheels, and a servo attached to steer its front wheels (Figure 33). The networked controls were simple: toggle front servo, toggle back DC motor controller, steer front servo left and right, gas and brake.

The RPI connected to the remote software that was run on a desktop PC in the same local area network. The PC had an Xbox 360 controller attached to it that was used to receive user input and send it to the RPI.

The RPI was running the software that controlled the peripherals. The RPI was connected to a remote desktop PC that was in the same local area network. The PC had an Xbox 360 controller attached to it that was used to receive user input and send it to the RPI. The PC screen displayed peripheral device statuses.

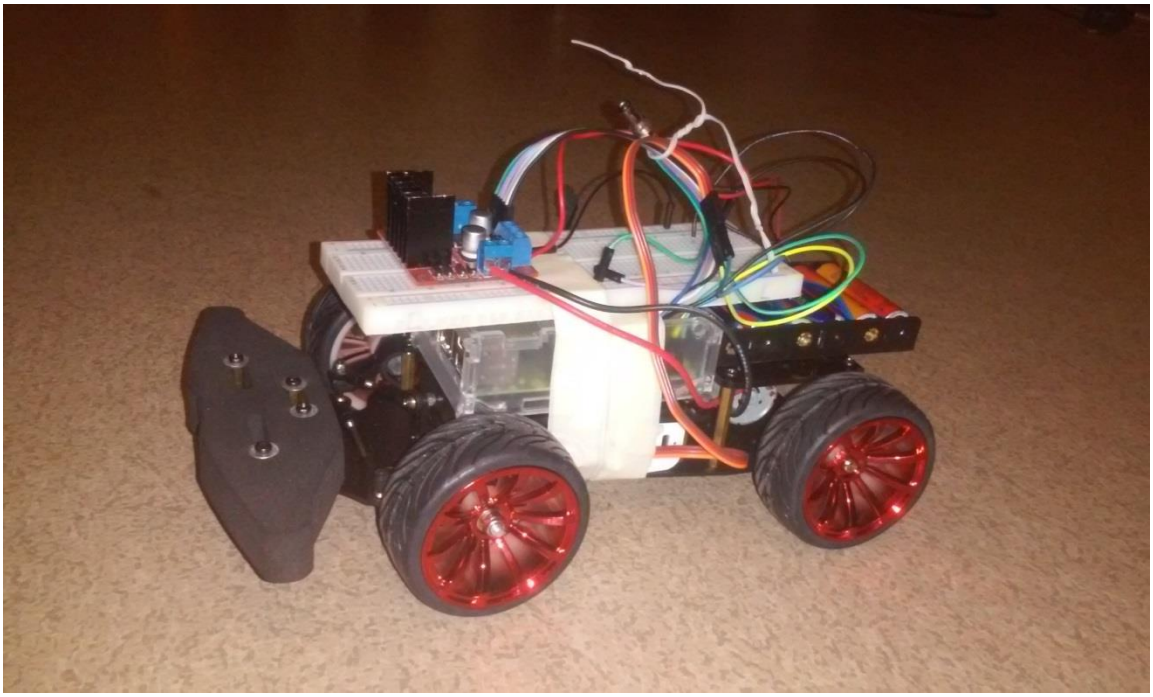

Figure 33. The RPi has a servo and a DC motor attached through the breadboard on the roof

4.7 SyncManager

Creating the network code for the car robot was cumbersome. Effort was put into considering how the process for creating new robots could be made easier. By inspecting the functionality of the servo class, a common observation was made. Both the RPI and the PC could have a similar servo class interface, each implementation being completely different. As an example, in both cases the servo class should have a function for setting the rotation of the servo motor. In the PC's case, this would queue up a networked instruction that should be passed on to the RPI. Once the RPI receives the rotation instruction, it calls upon its implementation of setting the rotation, which physically moves the servo motor.

The concept for a SyncManager class came to be. The idea being that the programmer defines a synchronizable type that completes the *SyncType* interface. That is, a set of functions that need to exist for the sync manager to be able to operate. This type must then have a remote counterpart. The counterpart's implementation mustn't be known, but it needs to be identifiable. Then, the programmer registers such type pairings into the SyncManager using the registerType() template function. Both the PC and the RPI code must register the synchronizable types accordingly, and if the type information matches, the two pieces of software should be able to communicate. Both the PC and the RPI must exchange their type information and confirm compatability before progressing any further. If the type exhange results in a success, the system is ready.

Figure 34 demonstrates sync manager use with a custom sync type *MyClass*. MyClass must be registered as a synchronizable type before the sync manager is initialized. Upon registering it, the three additional parameters must be provided: counter part name, id and version. These are required for confirming type compatibility.

```
codex::sync::Manager syncManager(socket);
syncManager.registerType<MyClass>(counterpartName, counterpartId,
counterpartVersion);
const bool ready = syncManager.initialize();
```

Figure 34. Sync manager creation and type registration

After initialization, the SyncManager can be used to instantiate local type instances (Figure 35). This will cause the remote site to instantiate a type that corresponds to the values of MyClass' counterpart name and id. Both the local as well as the remote instances' lifetimes are bound to the returned handle, which acts like a shared smart pointer.

```
codex::sync::Handle<MyClass> handle = syncManager.create<MyClass >();
if (handle)
    handle->myClassMethod();
```

Figure 35. Sync type instantiation

The first class to test the SyncManager's functionality was the original servo class. First, it had to be separated into two classes which I called ServoGhost and ServoShell. The ServoGhost is the PC side type, that sends the instructions. The ServoShell on the RPI receives these instructions and physically moves the servo motor.

Even after the creation of SyncManager, writing network code is still slow and error-prone. But it only needs to be done once per synchronizable type, saving a lot of time when reusing the same functionality.

Figure 36 displays a simplified flowchart explaining the three actors required to run the RPI client software. First, the matchmaking server is started. Next, both the desktop server as well as the RPI client connect to the matchmaking server. Their order of connection is irrelevant. The matchmaking server pairs them and they receive each other's endpoints. Sync manager type information is exchanged to confirm compatability. If the type information matches, they begin their designated program. In case that their type information is mismatching, each party has the chance to react without crashing the program.

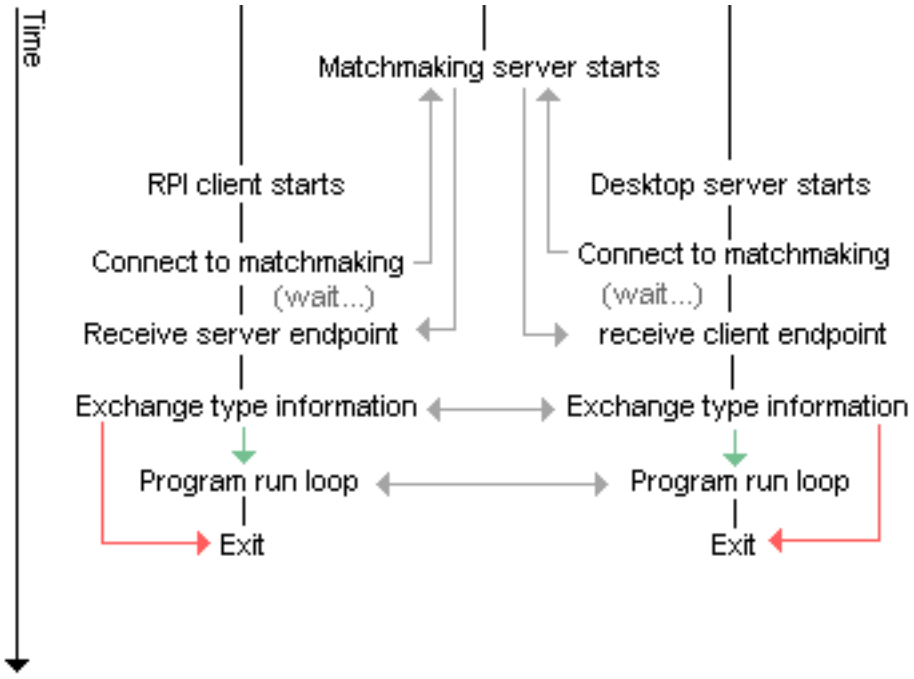

Figure 36. Program lifetime flowchart

4.8 SyncType

The SyncType interface (Figure 37) defines a set of member functions that can be overriden on will. The first syncUpdate function is used to signal the SyncManager that the SyncType instance is ready send an update sync packet. The other functions are sync packets that are sent in specific situations. syncCreate is called upon instantiating the object once. syncUpdate is called as requested. syncRemote is called when the instance is scheduled to be removed.

```
class ISyncType
{
public:
	typedef uint32_t SyncTypeVersionType;
	typedef std::string SyncTypeIdType;
public:
	virtual ~IType() {}

	/*
	Must implement the following static methods.
	static const std::string& getSyncTypeName();
	static const SyncTypeVersionType getSyncTypeVersion();
	*/

	/* By returning true, the sync type indicates
	that it is ready to write an update packet. */
	virtual bool syncUpdate(const spehs::time::Time deltaTime)
	{ return false; }

	/* Sync packets, optional implementation */
	virtual void syncCreate(protocol::WriteBuffer& buffer) {}
	virtual void syncCreate(protocol::ReadBuffer& buffer) {}
	virtual void syncUpdate(protocol::WriteBuffer& buffer) {}
	virtual void syncUpdate(protocol::ReadBuffer& buffer) {}
	virtual void syncRemove(protocol::WriteBuffer& buffer) {}
	virtual void syncRemove(protocol::ReadBuffer& buffer) {}
};
```

Figure 37. The SyncType interface

Each sync packet function has two versions, one with a WriteBuffer parameter and one with a ReadBuffer parameter. The write version is called for each locally managed SyncType instance. It allows the local type to write a message to the remote part, as correspondingly, the read version is used to receive the remote instance's message.

Two additional static functions must be implemented by the user. getSyncTypeName and getSyncTypeVersion. These functions are used in the type verification process when the two sync managers connect.

4.9 The final servo class

The final servo motor class (Figure 38) consists of a common interface class and two different implementations for it. The interface derives from the SyncType interface, but does not implement any of its virtual methods.

```
class IServo : public sync::IType
{
public:

	//Initialization
	/* Sets the pin that the servo is connected to. */
	virtual void setPin(const gpio::Pin pin) = 0;
	/* Set the minimum and maximum angle pulse durations. */
	virtual void setAngleLimits(
		const spehs::time::Time minPulseWidth,
		const spehs::time::Time maxPulseWidth,
		const float minAngle, const float maxAngle) = 0;
	virtual void setMinAngle(
		const spehs::time::Time minPulseWidth,
		const float minAngle) = 0;
	virtual void setMaxAngle(
		const spehs::time::Time maxPulseWidth,
		const float maxAngle) = 0;
	/* Set a rotation speed per second approximation. */
	virtual void setRotationSpeed(const float speed) = 0;

	//Control
	virtual void setActive(const bool isActive) = 0;
	virtual void setTargetAngle(const float angle) = 0;

	//State query
	virtual gpio::Pin getPin() const = 0;
	virtual float getApproximatedAngle() const = 0;
	virtual float getRotationSpeed() const = 0;
	virtual float getMinAngle() const = 0;
	virtual float getMaxAngle() const = 0;
};
```

Figure 38. The servo interface resembles the previous servo class

The servo interface declares a range of *pure* virtual methods. Pure virtual methods are virtual methods that are marked with the `= 0` postfix. All pure virtual methods must be overriden by the deriving class, or else the class will stays *abstract* and cannot be instantiated.

The servo ghost class (Figure 39) is a collection of synchronized servo settings (Figure 40), and an approximated value of the current servo motor angle. Modifying the *active* state or *targetAngle* enables the *syncRequested* boolean. When syncRequested is enabled, the *syncUpdate* method returns true, indicating that the servo ghost is ready to send an update packet to the servo shell. The update packet contains the updated values (Figure 41).

```
class ServoGhost final : public IServo
{
public:
	static const std::string& getSyncTypeName();
	static const SyncTypeIdType& getSyncTypeId();
	static const SyncTypeVersionType getSyncTypeVersion();
	ServoGhost();
	~ServoGhost();
	//Servo interface
	void setPin(const gpio::Pin pin) override;
	void setAngleLimits(
		const spehs::time::Time minPulseWidth,
		const spehs::time::Time maxPulseWidth,
		const float minAngle,
		const float maxAngle) override;
	void setMinAngle(
		const spehs::time::Time minPulseWidth,
		const float minAngle) override;
	void setMaxAngle(
		const spehs::time::Time maxPulseWidth,
		const float maxAngle) override;
	void setRotationSpeed(const float speed) override;
	void setActive(const bool isActive) override;
	void setTargetAngle(const float angle) override;
	gpio::Pin getPin() const override;
	float getApproximatedAngle() const override;
	float getRotationSpeed() const override;
	float getMinAngle() const override;
	float getMaxAngle() const override;
	//Sync type
	void syncCreate(protocol::WriteBuffer& buffer) override;
	void syncCreate(protocol::ReadBuffer& buffer) override;
	void syncRemove(protocol::WriteBuffer& buffer) override;
	void syncRemove(protocol::ReadBuffer& buffer) override;
	bool syncUpdate(const spehs::time::Time deltaTime) override;
	void syncUpdate(protocol::WriteBuffer& buffer) override;
	void syncUpdate(protocol::ReadBuffer& buffer) override;
private:
	float targetAngle;
	float approximatedAngle;
	gpio::Pin pin;
	spehs::time::Time minPulseWidth;
	spehs::time::Time maxPulseWidth;
	float rotationSpeed;
	float minAngle;
	float maxAngle;
	bool active;
	bool syncRequested;
};
```

Figure 39. The servo ghost class, used on the desktop PC software

```
void ServoGhost::syncCreate(protocol::WriteBuffer& buffer)
{
	buffer.write(pin);
	buffer.write(minPulseWidth);
	buffer.write(maxPulseWidth);
	buffer.write(rotationSpeed);
	buffer.write(minAngle);
	buffer.write(maxAngle);
}
```

Figure 40. Upon creation, the servo ghost synchronizes its settings to the servo shell

```
void ServoGhost::syncUpdate(protocol::WriteBuffer& buffer)
{
	buffer.write(active);
	buffer.write(targetAngle);
}

void ServoGhost::syncUpdate(protocol::ReadBuffer& buffer)
{
	buffer.read(approximatedAngle);
}
```

Figure 41. Servo ghost update sends requested state to the servo shell, and receives approximated angle values in return

The servo shell class (Figure 42) differs a lot from the servo ghost class. Most notably, it derives from a class called *ThreadedDevice*.

```
class ServoShell final : public ThreadedDevice, public IServo
{
public:
	static const std::string& getSyncTypeName();
	static const SyncTypeIdType& getSyncTypeId();
	static const SyncTypeVersionType getSyncTypeVersion();
	ServoShell();
	~ServoShell();
	//Servo interface
	void setPin(const gpio::Pin pin) override;
	void setAngleLimits(
		const spehs::time::Time minPulseWidth,
		const spehs::time::Time maxPulseWidth,
		const float minAngle,
		const float maxAngle) override;
	void setMinAngle(
		const spehs::time::Time minPulseWidth,
		const float minAngle) override;
	void setMaxAngle(
		const spehs::time::Time maxPulseWidth
		, const float maxAngle) override;
	void setRotationSpeed(const float speed) override;
	void setActive(const bool isActive) override;
	void setTargetAngle(const float angle) override;
	gpio::Pin getPin() const override;
	float getApproximatedAngle() const override;
	float getRotationSpeed() const override;
	float getMinAngle() const override;
	float getMaxAngle() const override;
	//Sync type
	void syncCreate(protocol::WriteBuffer& buffer) override;
	void syncCreate(protocol::ReadBuffer& buffer) override;
	void syncRemove(protocol::WriteBuffer& buffer) override;
	void syncRemove(protocol::ReadBuffer& buffer) override;
	bool syncUpdate(const spehs::time::Time deltaTime) override;
	void syncUpdate(protocol::WriteBuffer& buffer) override;
	void syncUpdate(protocol::ReadBuffer& buffer) override;
private:
	//Threaded device
	void onStart() override;
	void update() override;
	void onStop() override;
	mutable std::recursive_mutex mutex;
	spehs::time::Time lastUpdateTime;
	float targetAngle;
	float approximatedAngle;
	gpio::Pin pin;
	spehs::time::Time minPulseWidth;
	spehs::time::Time maxPulseWidth;
	float rotationSpeed;
	float minAngle;
	float maxAngle;
	bool active;
	spehs::time::Time updateTimer;
};
```

Figure 42. The servo shell class, used on the RPI software

The threaded device class is another interface that defines some common functionality between all of the devices that need an additional thread to run in. Threaded devices have the method *run* which launches the thread and calls the *onStart* method. The *update* method is called until the device is stopped. *onStop* method is called before the run thread exists. The multithreaded code is encapsulated within the scope of the class, thus all of its private data must be protected by a mutex.

Figure 43, the servo shell reads the update data sent by the servo ghost. Based on the received data, it can set its activity state and the target angle that it is being rotated towards. It sends back its currently approximated angle between set intervals. The physical servo motor doesn't provide any means for querying the state of the currently rotated angle, so this must be based on approximations.

```
void ServoShell::syncUpdate(protocol::ReadBuffer& buffer)
{
	bool _active;
	float _targetAngle;
	buffer.read(_active);
	buffer.read(_targetAngle);
	setActive(_active);
	setTargetAngle(_targetAngle);
}

void ServoShell::syncUpdate(protocol::WriteBuffer& buffer)
{
	std::lock_guard<std::recursive_mutex> lock(mutex);
	buffer.write(approximatedAngle);
}
```

Figure 43. The servo shell update

5 CONCLUSION

Creating any library in C++ is a lot of work. We, the programmers, come across many simple and elegant solutions in our daily lives. We should always be prepared to learn something new. The language is always evolving and our old habits can sometimes hold us back.

When beginning a large project like this, one should be certain that the development environment is sensible. Identifying any problems in the development pipeline is crucial. Creating tools can be beneficial if they can last.

From its start this thesis had numerous objectives and a loosely defined scope. The emphasis was shifted from robotics programming into general C++ programming with the addition of the networking elements. It manages to explain bits and pieces, here and there, but is no way a fully comprehensible guide to any specific subject. It can, however, contain some interesting, hopefully even creative ideas, that could be used in the context of creating a C++ robotics library from the scratch.

REFERENCES

[1]. Encyclopedia of computer science and technology, Henderson Harry. Revised edition. 2009.

[2] GPIO. Raspberry Pi documentation. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/gpio/README.md [Accessed 18.3.2018]

[3] Laplante, Philip A. (Dec 21, 2000). Dictionary of Computer Science, Engineering and Technology. CRC Press.

[4] The C++ programming language, Bjarne Stroustrup. 4th edition. 2013.

[5] Programming Paradigms, Ray Toal. Available: http://cs.lmu.edu/~ray/notes/paradigms/ [Accessed 18.3.2018]

[6] Functional programming introduction. Tutorials Point. Available: https://www.tutorialspoint.com/functional_programming/functional_programming_introduction.htm [Accessed 18.3.2018]

[7] Pragmatic Functional Programming, Robert C. Martin. The Clean Code Blog. 11th July 2017. Weblog. [Online] Available: https://blog.cleancoder.com/uncle-bob/2017/07/11/PragmaticFunctionalProgramming.html [Accessed 18.3.2018]

[8] Data oriented design and C++, Mike Acton. CppCon 2014. [Online] Available: https://www.youtube.com/watch?v=rX0ItVEVjHc [Accessed 25.2.2018]

[9] Boost libraries. [Online] Available: http://www.boost.org/ [Accessed 22.3.2018]

[10] CppReference. [Online] Available: http://en.cppreference.com/w/ [Accessed 22.3.2018]

[11] SpehsEngine. Available: https://github.com/Yuuso/SpehsEngine [Accessed 18.3.2018]

[12] Compilers - What Every Programmer Should Know About Compiler Optimizations, Hadi Brais. Available: https://msdn.microsoft.com/en-us/magazine/dn904673.aspx [Accessed: 18.3.2018]

[13] C++ References, Alex Allain. Available: https://www.cprogramming.com/tutorial/references.html [Accessed: 18.3.2018]

[14] A foolish consistency, Jon Kalb. Weblog. [Online] Available: http://slashslash.info/2018/02/a-foolish-consistency/ [Accessed 23.3.2018]

[15] Boccara, J. Strong types for strong interfaces. Fluent C++. Weblog. [Online] Available: https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/ [Accessed 26 February 2018].

[16] C++ Templates. Tutorials Point. Available: https://www.tutorialspoint.com/cplusplus/cpp_templates.htm [Accessed 18.3.2018]

[17] The stack and the heap. LearnCpp.com. Available: http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/ [Accessed 18.3.2018]

[18] Raspberry Pi homepage. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ [Accessed December 30th 2017]

[19] Arduino store. Available: https://store.arduino.cc/arduino-uno-rev3 [Accessed February 27th 2018]

[20] Raspberry Pi operating system. Available: https://www.raspberrypi.org/downloads/raspbian/ [Accessed 19.3.2018]

[21] Visual Studio for Linux development. Available: https://blogs.msdn.microsoft.com/vcblog/2017/04/11/linux-development-with-c-in-visual-studio/ [Accessed 19.3.2018]